Parte 2

Il linguaggio Java

IN QUESTA PARTE

CAPITOLO 3

Espressioni, operatori e strutture di controllo

di Michael Morrison

IN QUESTO CAPITOLO

Nel Capitolo 3 sono stati discussi alcuni componenti fondamentali dei programmi Java. Questo capitolo si concentra sull'utilizzo di questi componenti per eseguire operazioni più utili; i tipi di dati sono interessanti, ma senza le espressioni e gli operatori, non è possibile farci granché. Tuttavia, anche le espressioni e gli operatori da soli sono limitati in ciò che fanno; utilizzando le strutture di controllo si ha la possibilità di fare cose più interessanti.

In questo capitolo vengono discussi tutti questi argomenti e, come in un puzzle, vengono riunite molte parti della programmazione in Java che si è iniziato ad analizzare. Ciò consente non solo di migliorare la conoscenza del linguaggio Java, ma anche di apprendere che cosa serve per scrivere programmi più interessanti.

Espressioni e operatori

Dopo aver creato una variabile, di norma si desidera utilizzarla a qualche scopo. Gli *operatori* permettono di eseguire una valutazione o un calcolo su uno o più oggetti; applicati a *variabili* e *letterali*, formano le *espressioni*. Si può pensare alle espressioni come a equazioni programmatiche; in modo più formale, si può dire che sono sequenze di uno o più oggetti dati (operandi) e zero o più operatori che producono un risultato. Un esempio di espressione può essere il seguente:

```
x = y / 3
```

In questa espressione, x e y sono variabili, 3 è un letterale, = e / sono operatori. Questa espressione indica che la variabile y è divisa per 3 utilizzando l'operatore di divisione (/) e il risultato è memorizzato in x utilizzando l'operatore di assegnamento (=). Si noti che l'espressione è stata descritta da destra a sinistra. Nonostante questo approccio all'analisi di un'espressione sia utile per descrivere l'operazione di assegnamento, in realtà la maggior parte delle espressioni in Java viene valutata da sinistra a destra, come viene spiegato meglio nel prossimo paragrafo.

Precedenza tra gli operatori

Nonostante le espressioni di Java vengano di norma valutate da sinistra a destra, se non vi fossero altre regole, in molti casi non sarebbe possibile determinare il risultato di un'espressione. La seguente espressione evidenzia il problema:

```
x = 2 * 6 + 16 / 4
```

Se si utilizzasse solamente la valutazione da sinistra a destra, l'operazione di moltiplicazione 2 * 6 verrebbe eseguita per prima, dando come risultato di 12. Di seguito verrebbero eseguite l'operazione di addizione 12 + 16, che dà come risultato 28, e quindi l'operazione di divisione 28 / 4, che dà come risultato 7. Infine verrebbe eseguita l'operazione di assegnamento x = 7, nella quale il numero 7 viene assegnano alla variabile x.

Chi ha esperienza con la *precedenza tra gli operatori* in un altro linguaggio, probabilmente si è accorto che la valutazione di questa espressione è errata. Il problema consiste nel fatto che l'utilizzo della semplice valutazione da sinistra a destra di un'espressione può produrre risultati errati, a seconda dell'ordine degli operatori. La soluzione viene offerta proprio dalla *precedenza tra gli operatori*, che determina l'ordine in cui questi vengono valutati. In Java, a ogni operatore è associato un ordine di precedenza. Di seguito è riportato un elenco di tutti gli operatori di Java; il livello di precedenza diminuisce riga per riga: gli operatori di una riga hanno precedenza maggiore rispetto a quelli della riga successiva, mentre gli operatori sulla stessa riga hanno tutti uguale precedenza. Ad esempio l'operatore [] ha precedenza più alta rispetto all'operatore *, ma ha la stessa precedenza dell'operatore ().

```
.      []     ()
++     --     !      ~
*      /      %
+      -
<<     >>     >>>
<      >      <=     >=
==     !=
&
^
&&
||
?:
=
```

La valutazione delle espressioni procede sempre da sinistra a destra, ma solo per gli operatori con la stessa precedenza. Diversamente, gli operatori con precedenza maggiore vengono valutati prima di quelli con precedenza inferiore.

Sulla base di ciò, si osservi una semplice equazione:

```
x = 2 * 6 + 16 / 4
```

Prima di iniziare la valutazione da sinistra a destra, è necessario controllare se vi sono operatori con precedenza diversa. In questo caso, gli operatori di moltiplicazione (*) e di divisione (/) hanno la precedenza più alta, seguiti dall'operatore di addizione (+) e quindi dall'operatore di assegnamento (=). Poiché gli operatori di moltiplicazione e di divisione hanno la stessa precedenza, devono essere valutati da sinistra a destra. Si esegue per prima l'operazione di moltiplicazione `2 * 6`, che dà come risultato `12`, quindi si esegue la divisione `16 / 4`, che dà come risultato `4`. Dopo aver eseguito queste due operazioni, l'espressione risulta essere:

```
x = 12 + 4;
```

Poiché l'operatore di addizione ha una precedenza più alta rispetto all'operatore di assegnamento, viene eseguita l'operazione di addizione `12 + 4`, che dà come risultato `16`, e infine l'operazione di assegnamento, che assegna il numero `16` alla variabile `x`. Come si può vedere, se si valuta l'espressione utilizzando la precedenza tra gli operatori, si ottiene un risultato diverso.

Per capire meglio questo concetto, si osservi un'altra espressione che utilizza le parentesi:

```
x = 2 * (11 - 7);
```

Senza le parentesi, verrebbe eseguita prima la moltiplicazione, quindi la sottrazione. Tuttavia, facendo riferimento all'elenco di precedenze, l'operatore `()` viene prima di tutti gli altri e pertanto viene eseguita per prima l'operazione `11 – 7`, che dà come risultato `4`. L'espressione pertanto diventa:

```
x = 2 * 4;
```

Il resto dell'espressione viene facilmente risolto con una moltiplicazione e un'operazione di assegnamento per ottenere un risultato di `8` nella variabile `x`.

Operatori interi

Vi sono tre tipi di operazioni che possono essere eseguite sugli interi: unari, binari e relazionali. Gli operatori unari agiscono solo su numeri interi singoli, mentre quelli binari agiscono su coppie di numeri interi. Entrambi questi operatori di norma restituiscono un numero intero. Gli operatori relazionali, invece, agiscono su due numeri interi, ma restituiscono un risultato booleano, anziché un intero.

Gli operatori interi unari e binari di norma restituiscono un tipo `int`. Per tutte le operazioni che includono i tipi `byte`, `short` e `int`, il risultato è sempre un `int`. L'unica eccezione si ha quando uno degli operatori è un `long`, nel qual caso anche il risultato sarà di tipo `long`.

Operatori interi unari

Gli operatori interi unari, elencati nella Tabella 3.1, agiscono su un singolo numero intero.

Tabella 3.1 *Gli operatori interi unari.*

Descrizione	*Operatore*
Incremento	++
Decremento	--
Negazione	-
Complemento bit per bit	~

Gli operatori ++ e -- incrementano e decrementano di 1 le variabili di tipo intero. Come in C e in C++, questi operatori possono essere prefissi o postfissi. Un *operatore prefisso* ha effetto prima della valutazione dell'espressione in cui si trova, mentre un *operatore postfisso* ha effetto dopo che è stata valutata l'espressione; gli operatori unari postfissi vengono inseriti immediatamente dopo la variabile. Di seguito viene mostrato un esempio di ciascun tipo di operatore:

```
y = ++x;
z = x–;
```

Nel primo esempio, `x` è *incrementato prefisso*, vale a dire che viene incrementato prima di essere assegnato a `y`. Nel secondo esempio, `x` è *decrementato postfisso*, vale a dire che viene decrementato dopo essere stato assegnato a `z`, cioè il valore di `x` viene assegnato a `z` prima di essere decrementato. Il Listato 3.1 include il programma `IncDec`, che utilizza entrambi i tipi di operatori. Si noti che questo programma viene implementato nella classe Java `IncDec`. Questo è il risultato della struttura orientata agli oggetti di Java, che richiede che i programmi siano implementati come classi. Quando si vede un riferimento a un *programma* Java, è opportuno tenere a mente che in realtà si fa riferimento a una *classe* Java.

Listato 3.1 *La classe IncDec.*

```
class IncDec {
  public static void main (String args[]) {
    int x = 8, y = 13;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("++x = " + ++x);
    System.out.println("y++ = " + y++);
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

Il programma `IncDec` produce il seguente risultato:

```
x = 8
y = 13
++x = 9
y++ = 13
x = 9
y = 14
```

L'operatore intero unario di negazione (-) viene utilizzato per modificare il segno di un valore intero. Questo operatore è molto semplice, come si può vedere di seguito:

```
x = 8;
y = -x;
```

In questo esempio, a x viene assegnato il valore letterale 8, che quindi viene negato e assegnato a y. Il valore risultante di y è -8. Per vedere questo codice in un vero programma Java, si osservi il programma Negazione nel Listato 3.2.

Listato 3.2 *La classe Negazione.*

```
class Negazione {
  public static void main (String args[]) {
    int x = 8;
    System.out.println("x = " + x);
    int y = -x;
    System.out.println("y = " + y);
  }
}
```

L'ultimo operatore intero unario di Java è l'*operatore di complemento bit per bit* (~), che esegue una negazione bit per bit di un valore intero. *Negazione bit per bit* significa che viene invertito ogni bit nel numero: in altre parole, tutti gli 0 binari diventano 1 e tutti gli 1 diventano 0. Ecco un esempio molto simile a quello per l'operatore di negazione:

```
x = 8;
y = ~x;
```

Anche in questo esempio a x viene assegnato il valore letterale 8, ma prima dell'assegnamento a a y viene eseguita l'operazione di complemento bit per bit. Senza scendere nel dettaglio di come vengono memorizzati i numeri interi, ciò significa che vengono modificati tutti i bit della variabile x, dando un risultato decimale uguale a -9. Questo risultato è determinato dal fatto che i numeri negativi vengono memorizzati utilizzando un metodo noto come *complemento a due*. Ciò può essere verificato con il programma ComplementoBitPerBit, incluso nel Listato 3.3.

I numeri interi sono memorizzati come serie di bit binari, ognuno dei quali può avere un valore 0 o 1. Un numero viene considerato negativo se il bit di ordine maggiore è 1. Poiché un complemento bit per bit modifica tutti i bit di un numero, incluso quello di ordine maggiore, vengono invertiti il segno e il numero.

Listato 3.3 *La classe ComplementoBitPerBit.*

```
class ComplementoBitPerBit {
  public static void main (String args[]) {
    int x = 8;
    System.out.println("x = " + x);
    int y = ~x;
    System.out.println("y = " + y);
  }
}
```

Operatori interi binari

Gli operatori interi binari, elencati nella Tabella 3.2, agiscono su coppie di interi.

Tabella 3.2 *Gli operatori interi binari.*

Descrizione	*Operatore*
Addizione	+
Sottrazione	-
Moltiplicazione	*
Divisione	/
Modulo	%
AND bit per bit	&
OR bit per bit	¦
XOR bit per bit	^
Scorrimento a sinistra	<<
Scorrimento a destra	>>
Scorrimento a destra con riempimento per zeri	>>>

Gli operatori di addizione, di sottrazione, di moltiplicazione e di divisione (+, -, *, e /) si comportano come previsto.

Si noti il funzionamento dell'operatore di divisione: poiché si opera con operandi interi, restituisce un intero. Nei casi in cui il risultato dell'operazione prevede un resto, è possibile utilizzare l'operatore di modulo (%) per ottenerne il valore.

Il Listato 3.4 include il programma `Aritmetica`, che mostra come funzionano gli operatori aritmetici interi binari di base.

Listato 3.4 *La classe Aritmetica.*

```
class Aritmetica {

  public static void main (String args[]) {
    int x = 17, y = 5;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("x + y = " + (x + y));
    System.out.println("x - y = " + (x - y));
    System.out.println("x * y = " + (x * y));
    System.out.println("x / y = " + (x / y));
    System.out.println("x % y = " + (x % y));
  }

}
```

Il risultato dell'esecuzione del programma `Aritmetica` è:

```
x = 17
y = 5
x + y = 22
x - y = 12
x * y = 85
x / y = 3
x % y = 2
```

Questi risultati non dovrebbero sorprendere. Si noti solamente che l'operazione di divisione `x / y`, in questo caso `17 / 5`, dà come risultato `3` e che l'operazione di modulo `x % y`, che dipende da `17 % 5`, dà come risultato `2` (il resto della divisione dei numeri interi).

Dal punto di vista matematico, la divisione per zero dà come risultato un numero infinito. Poiché per i computer è molto difficile rappresentare i numeri infiniti, il risultato di un'operazione di divisione o di modulo per zero è un errore, vale a dire che viene creata un'eccezione di esecuzione. Le eccezioni verranno discusse nel Capitolo 6.

Gli operatori `AND`, `OR` e `XOR` bit per bit (`&`, `¦`, e `^`) agiscono tutti sui singoli bit di un intero. Risultano utili a volte quando un numero intero viene utilizzato come campo di bit. Se ne può vedere un esempio quando viene utilizzato un intero per rappresentare un gruppo di flag binari. Un `int` è in grado di rappresentare fino a 32 flag diversi, in quanto è memorizzato in 32 bit. Il Listato 3.5 include il programma `BitPerBit`, che mostra come utilizzare gli operatori interi binari bit per bit.

In Java è inclusa una classe che fornisce supporto specificamente per memorizzare i flag binari, chiamata BitSet, descritta nel Capitolo 10.

Listato 3.5 *La classe BitPerBit.*

```
class BitPerBit {
  public static void main (String args[]) {
    int x = 5, y = 6;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("x & y = " + (x & y));
    System.out.println("x ¦ y = " + (x ¦ y));
    System.out.println("x ^ y = " + (x ^ y));
  }
}
```

L'output dell'esecuzione di `BitPerBit` è:

```
x = 5
y = 6
x & y = 4
x ¦ y = 7
x ^ y = 3
```

Per capire questo output, è necessario capire prima gli equivalenti binari di ogni numero decimale. In `BitPerBit`, le variabili `x` e `y` sono impostate su `5` e `6`, che corrispondono ai

numeri binari 0101 e 0110. L'operazione AND bit per bit confronta ogni bit di ogni numero per vedere se coincidono, quindi imposta il bit risultante su 1, se entrambi i bit confrontati sono 1, diversamente su 0. Il risultato dell'operazione AND bit per bit su questi due numeri è 0100 in binario, o 4 decimale. La stessa logica viene utilizzata per entrambi gli altri operatori, per i quali variano solamente le regole per il confronto dei bit. L'operatore OR bit per bit imposta il bit risultante su 1 se nessuno dei bit confrontati è 1. Per questi numeri, il risultato è 0111 binario, o 7 decimale; l'operatore XOR bit per bit imposta i bit risultanti su 1 se uno solo dei bit confrontati è 1, diversamente su 0. Per questi numeri il risultato è 0011 binario o 3 decimale.

Gli operatori di scorrimento a sinistra, di scorrimento a destra e di scorrimento a destra con riempimento per zeri (<<, >>, e >>>) fanno scorrere i singoli bit di un intero in base a un valore intero determinato. Di seguito vengono riportati alcuni esempi di come vengono utilizzati questi operatori:

```
x << 3;
y >> 7;
z >>> 2;
```

Nel primo esempio, i singoli bit della variabile intera x vengono fatti scorrere a sinistra di tre posti. Nel secondo esempio, i bit di y vengono fatti scorrere a destra di 7 posti. Infine, nel terzo esempio, z viene fatto scorrere a destra di due posti, muovendo gli zeri nei due posti di sinistra. Per vedere gli operatori di scorrimento in un programma vero, si osservi il Listato 3.6.

Listato 3.6 *La classe Scorrimento.*

```
class Scorrimento {
  public static void main (String args[]) {
    int x = 7;
    System.out.println("x = " + x);
    System.out.println("x >> 2 = " + (x >> 2));
    System.out.println("x << 1 = " + (x << 1));
    System.out.println("x >>> 1 = " + (x >>> 1));
  }
}
```

L'output di Scorrimento è:

```
x = 7
x >> 2 = 1
x << 1 = 14
x >>> 1 = 3
```

La cifra che viene fatta scorrere in questo caso è il decimale 7, che in binario viene rappresentato come 0111. La prima operazione di scorrimento a destra fa scorrere i bit due posti a destra, dando come risultato il numero binario 0001, o 1 decimale. L'operazione successiva, uno scorrimento a sinistra, fa scorrere i bit di un posto verso sinistra, dando come risultato il numero binario 1110 o 14 decimale. L'ultima operazione è uno scorrimento a destra con riempimento per zeri, che fa scorrere i bit di un posto verso destra, dando come risultato il numero binario 0011 o 3 decimale.

Sulla base di questi esempi, ci si potrebbe chiedere quale sia la differenza tra gli operatori di scorrimento a destra (>>) e di scorrimento a destra con riempimento per zeri (>>>). Apparentemente l'operatore di scorrimento a destra fa scorrere gli zeri nei bit di sinistra, esattamente come l'operatore di scorrimento a destra con riempimento per zeri. Quando si agisce su numeri positivi, tra i due operatori non vi è differenza, in quanto entrambi fanno scorrere gli zeri nei bit superiori di un numero. La differenza sorge quando si fanno scorrere numeri negativi, il cui bit di ordine maggiore è impostato su 1. L'operatore di scorrimento a destra mantiene il bit di ordine superiore e in realtà fa scorrere a destra i 31 bit di ordine inferiore. Questo comportamento permette di ottenere risultati per i numeri negativi simili a quelli per i numeri positivi. Ad esempio, `-8` fatto scorrere a destra di un posto dà `-4`. L'operatore di scorrimento a destra con riempimento per zeri, invece, sposta gli zeri in tutti i bit superiori, incluso il bit di ordine maggiore. Quando si applica questo scorrimento ai numeri negativi, il bit di ordine maggiore diventa `0` e il numero diventa positivo.

Operatori interi relazionali

L'ultimo gruppo di operatori interi è composto dagli operatori relazionali, elencati nella Tabella 3.3, che operano sui numeri interi ma restituiscono un tipo `boolean`.

Tabella 3.3 *Gli operatori interi relazionali.*

Descrizione	*Operatore*
Minore di	<
Maggiore di	>
Minore di o uguale a	<=
Maggiore di o uguale a	>=
Uguale a	==
Diverso da	!=

Tutti questi operatori eseguono dei confronti tra numeri interi. Il Listato 3.7 include il programma `Relazionali`, che mostra l'utilizzo degli operatori interi relazionali.

Listato 3.7 *La classe Relazionali.*

```
class Relazionali {
  public static void main (String args[]) {
    int x = 7, y = 11, z = 11;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("z = " + z);
    System.out.println("x < y = " + (x < y));
    System.out.println("x > z = " + (x > z));
    System.out.println("y <= z = " + (y <= z));
    System.out.println("x >= y = " + (x >= y));
```

```
        System.out.println("y == z = " + (y == z));
        System.out.println("x != y = " + (x != z));
    }
}
```

L'output è il seguente:

```
x = 7
y = 11
z = 11
x < y = true
x > z = false
y <= z = true
x >= y = false
y == z = true
x != y = true
```

Come si può vedere, il metodo `println()` permette di stampare correttamente risultati booleani come `true` e `false`.

Operatori in virgola mobile

Come per gli operatori interi, vi sono tre tipi di operazioni che possono essere eseguite sui numeri in virgola mobile: unarie, binarie e relazionali. Gli operatori unari agiscono solo su numeri in virgola mobile singoli, mentre gli operatori binari agiscono su coppie di numeri in virgola mobile. Entrambi questi operatori restituiscono un risultato in virgola mobile. Gli operatori relazionali, invece, agiscono su due numeri in virgola mobile, ma restituiscono un risultato booleano.

Gli operatori in virgola mobile unari e binari restituiscono un tipo `float`, se entrambi gli operandi sono di tipo `float`. Se uno o entrambi gli operandi sono di tipo `double`, il risultato dell'operazione è di tipo `double`.

Operatori in virgola mobile unari

Gli operatori in virgola mobile unari, elencati nella Tabella 3.4, agiscono su un singolo numero in virgola mobile.

Tabella 3.4 *Gli operatori in virgola mobile unari.*

Descrizione	*Operatore*
Incremento	++
Decremento	--

Come si può vedere, gli unici due operatori in virgola mobile unari sono gli operatori di incremento e decremento, che rispettivamente aggiungono e sottraggono `1.0` dagli operandi in virgola mobile.

Operatori in virgola mobile binari

Gli operatori in virgola mobile binari, elencati nella Tabella 3.5, agiscono su una coppia di numeri in virgola mobile.

Tabella 3.5 *Gli operatori in virgola mobile binari.*

Descrizione	*Operatore*
Addizione	+
Sottrazione	-
Moltiplicazione	*
Divisione	/
Modulo	%

Gli operatori in virgola mobile binari sono costituiti dalle quattro operazioni binarie tradizionali (+, -, *, /) e dall'operatore di modulo (%). Ci si potrebbe chiedere come l'operatore di modulo rientri tra questi, considerato che il suo utilizzo come operatore intero si basa sulla divisione intera. L'operatore intero di modulo restituisce il resto di una divisione intera dei due operandi, ma una divisione in virgola mobile non ha mai resto. L'operatore modulo in virgola mobile restituisce l'equivalente in virgola mobile di una divisione a numeri interi. Ciò significa che la divisione viene eseguita con entrambi gli operandi in virgola mobile, ma il divisore viene trattato come numero intero, cosa che produce un resto in virgola mobile. Il Listato 3.8 contiene il programma `VirgolaMobile`, che mostra il funzionamento dell'operatore di modulo in virgola mobile assieme agli altri operatori in virgola mobile binari.

Listato 3.8 *La classe VirgolaMobile.*

```
class VirgolaMobile {
  public static void main (String args[]) {
    float x = 23.5F, y = 7.3F;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("x + y = " + (x + y));
    System.out.println("x - y = " + (x - y));
    System.out.println("x * y = " + (x * y));
    System.out.println("x / y = " + (x / y));
    System.out.println("x % y = " + (x % y));
  }
}
```

L'output di `VirgolaMobile` è:

```
x = 23.5
y = 7.3
x + y = 30.8
x - y = 16.2
```

```
x * y = 171.55
x / y = 3.21918
x % y = 1.6
```

Le prime quattro operazioni hanno prodotto il risultato previsto, partendo dai due operandi in virgola mobile e producendo un risultato in virgola mobile. L'operazione di modulo finale ha determinato che 23.5 diviso 7.3 dà un risultato intero di 3, con un resto di 1.6.

Operatori in virgola mobile relazionali

Gli operatori in virgola mobile relazionali confrontano due operandi in virgola mobile e producono un risultato booleano. Sono uguali agli operatori interi relazionali elencati nella precedente Tabella 3.3, a eccezione del fatto che operano su numeri in virgola mobile.

Operatori booleani

Gli operatori booleani, elencati nella Tabella 3.6, agiscono su tipi booleani e restituiscono un risultato booleano.

Tabella 3.6 *Gli operatori booleani.*

Descrizione	*Operatore*
AND di valutazione	&
OR di valutazione	|
XOR di valutazione	^
AND logico	&&
OR logico	||
Negazione	!
Uguale a	==
Diverso da	!=
Condizionale	?:

Gli operatori di valutazione (&, |, e ^) valutano entrambi i membri di un'espressione prima di determinarne il risultato. Gli operatori logici (&& e ||) non valutano la parte destra dell'espressione, se non è necessario. Per capire meglio la differenza tra questi operatori, si osservino le due espressioni seguenti:

```
boolean risultato = isValid & (Conta > 10);
boolean risultato = isValid && (Conta > 10);
```

La prima espressione utilizza l'operatore AND di valutazione (&) per effettuare un assegnamento. In questo caso, vengono sempre valutati entrambi i membri dell'espressione, indipendentemente dai valori delle variabili interessate. Nella seconda espressione viene utilizzato l'operatore AND logico (&&).

Questa volta viene prima controllato il valore booleano `isValid`; se questo è `false`, il membro destro dell'espressione viene ignorato e viene eseguito l'assegnamento. Questo operatore è più efficiente, in quanto un valore `false` nella parte sinistra dell'espressione fornisce sufficienti informazioni per determinare il risultato `false`.

Nonostante gli operatori logici siano più efficienti degli operatori di valutazione, a volte è preferibile utilizzare questi ultimi per assicurarsi che venga valutata l'intera espressione. Il seguente codice mostra come l'operatore `AND` di valutazione sia necessario per valutare completamente un'espressione.

```
while ((++x < 10) & (++y < 15)) {
  System.out.println(x);
  System.out.println(y);
}
```

In questo esempio, la seconda espressione (`++y < 15`) viene valutata dopo l'ultimo passaggio del ciclo a causa dell'operatore `AND` di valutazione. Se fosse stato utilizzato l'operatore `AND` logico, la seconda espressione non sarebbe stata valutata e `y` non sarebbe stato incrementato dopo l'ultimo passaggio.

I tre operatori booleani, negazione, uguaglianza e diverso da (`!`, `==`, e `!=`), si comportano esattamente come previsto. L'operatore di negazione modifica un valore booleano da `false` a `true` o da `true` a `false`, a seconda del valore originale, l'operatore di uguaglianza determina semplicemente se due valori booleani sono uguali (entrambi `true` o entrambi `false`), in modo simile l'operatore "diverso da" determina se due operandi booleani sono differenti.

L'operatore booleano condizionale (`?:`) è il più particolare degli operatori booleani e vale la pena dargli un'occhiata più da vicino. Questo operatore è detto anche *ternario*, in quanto utilizza tre elementi: una condizione e due espressioni. La sintassi è:

```
Condizione ? Espressione1 : Espressione2
```

La *`Condizione`*, che è essa stessa booleana, viene valutata per prima per determinare se sia `true` o `false`; se viene valutata come `true`, viene valutata la *`Espressione1`*, se viene valutata come `false`, viene valutata la *`Espressione2`*. Per avere un'idea più chiara dell'operatore condizionale, si osservi il Listato 3.9.

Listato 3.9 *La classe Condizionale.*

```
class Condizionale {
  public static void main (String args[]) {
    int x = 0;
    boolean seiPari = false;
    System.out.println("x = " + x);
    x = seiPari ? 4 : 7;
    System.out.println("x = " + x);
  }
}
```

Il risultato del programma `Condizionale` è:

```
x = 0
x = 7
```

Alla variabile intera `x` viene prima assegnato il valore `0`, mentre alla variabile booleana `seiPari` viene assegnato il valore `false`. Utilizzando l'operatore condizionale, viene controllato il valore di `seiPari`; poiché questo è `false`, viene utilizzata la seconda espressione della condizione e a `x` viene assegnato il valore `7`.

Operatore su stringhe

Esattamente come i numeri interi, i numeri in virgola mobile e i booleani, anche le stringhe possono essere manipolate con gli operatori. In realtà esiste un solo operatore che agisce sulle stringhe: l'operatore di concatenazione (+). Come indicato nel programma `Concatenazione` incluso nel Listato 3.10, questo funziona in modo molto simile all'operatore di addizione dei numeri, vale a dire unisce le stringhe.

Listato 3.10 *La classe Concatenazione.*

```
class Concatenazione {
  public static void main (String args[]) {
    String primaMeta = "Che " + "cosa ";
    String secondaMeta = "vuoi " + "dire?";
    System.out.println(primaMeta + secondaMeta);
  }
}
```

L'output di `Concatenazione` è:

```
Che cosa vuoi dire?
```

Nel programma `Concatenazione`, i letterali stringa vengono concatenati per effettuare assegnamenti alle due variabili stringa, `primaMeta` e `secondaMeta`, al momento della creazione. Le due variabili stringa vengono quindi concatenate all'interno della chiamata al metodo `println()`.

Operatori di assegnamento

L'ultimo gruppo di operatori non ancora esaminati è quello degli operatori di assegnamento, elencati nella Tabella 3.7, che funzionano con tutti i tipi di dati primitivi.

A eccezione dell'operatore di assegnamento semplice (=), gli altri funzionano esattamente come le rispettive controparti che non eseguono l'assegnamento, con l'unica differenza che il risultato viene memorizzato nell'operando di sinistra. Si osservino i seguenti esempi:

```
x += 6;
x *= (y - 3);
```

Nel primo esempio vengono sommati x e 6 e il risultato viene memorizzato in x. Nel secondo esempio da y viene sottratto 3 e il risultato viene moltiplicato per x. Il risultato finale viene quindi memorizzato in x.

Tabella 3.7 *Gli operatori di assegnamento.*

Descrizione	*Operatore*
Semplice	=
Addizione	+=
Sottrazione	-=
Moltiplicazione	*=
Divisione	/=
Modulo	%=
AND	&=
OR	|=
XOR	^=

Strutture di controllo

Nonostante le operazioni sui dati siano molto utili, è ora di passare a un nuovo argomento: il controllo del flusso del programma. Il flusso di un programma viene determinato da due tipi diversi di costrutti: le diramazioni e i cicli. Le *diramazioni* permettono di selezionare la parte di un programma da eseguire, i *cicli* invece forniscono uno strumento per ripetere determinate parti di un programma. Insieme, le diramazioni e i cicli offrono un mezzo potente per controllare la logica e l'esecuzione del codice.

Diramazioni

Senza diramazioni o cicli, il codice Java viene eseguito in modo sequenziale, come indicato nella Figura 3.1. In questa figura, le istruzioni vengono eseguite in modo sequenziale, ma se non si desidera che venga eseguita ogni singola istruzione, è necessario utilizzare una diramazione. Nella Figura 3.2 è mostrato come una diramazione condizionale permette di modificare il flusso del codice.

Aggiungendo una diramazione si offrono al codice due diverse strade da intraprendere, in base al risultato dell'espressione condizionale. Il concetto di diramazione potrebbe sembrare banale, ma sarebbe difficile, se non impossibile, scrivere programmi utili senza di esse. In Java sono supportati due tipi di diramazioni: `if-else` e `switch`.

Figura 3.1
Un programma eseguito in modo sequenziale.

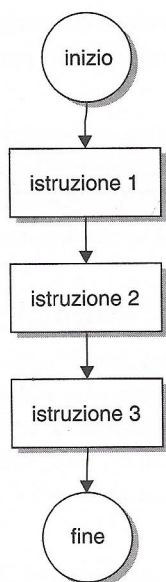
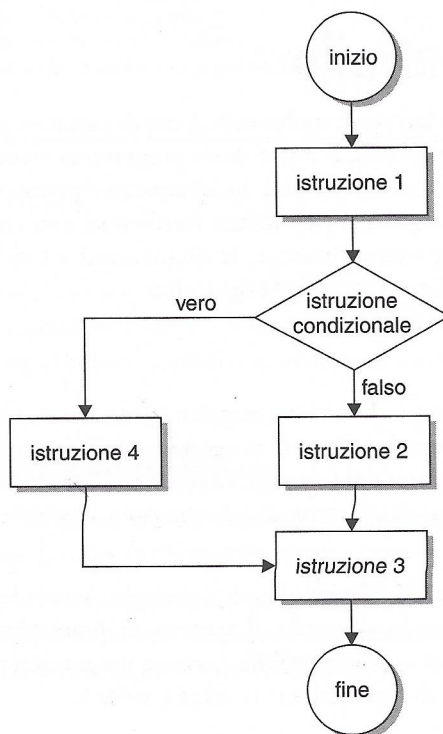



Figura 3.2
Un programma con una diramazione.

Diramazioni if-else

Le diramazioni `if-else` sono quelle utilizzate più comunemente nella programmazione Java. Vengono utilizzate per selezionare in modo condizionale uno fra due risultati. La sintassi per l'istruzione `if-else` è:

```
if (Condizione)
  Istruzione1
else
  Istruzione2
```

Se la *Condizione* booleana viene valutata come `true`, viene eseguita la *Istruzione1*. In modo simile, se la *Condizione* viene valutata come `false`, viene eseguita la *Istruzione2*. Il secondo esempio mostra l'utilizzo di un'istruzione `if-else`:

```
if (sonoStanco)
  devoMangiare = true;
else
  devoMangiare = false;
```

Se la variabile booleana `sonoStanco` restituisce `true`, viene eseguita la prima istruzione e `devoMangare` viene impostata su `true`. Diversamente, viene eseguita la seconda istruzione e `devoMangare` viene impostata su `false`. La diramazione `if-else` funziona in modo molto simile all'operatore condizionale (`?:`) descritto precedentemente in questo capitolo e in realtà è possibile considerarla come una versione ampliata di tale operatore. Una differenza significativa è che nelle diramazioni `if-else` è possibile includere istruzioni composte, cosa che non è possibile fare con gli operatori condizionali.

Le istruzioni composte sono blocchi di codice inclusi tra parentesi graffe {} che appaiono a un blocco di codice esterno come una singola istruzione.

Se si ha una sola istruzione da eseguire in modo condizionale, è possibile tralasciare la parte `else`, come indicato nel seguente esempio:

```
if (sonoAssetato)
  devoBere = true;
```

Se invece sono necessari più di due risultati condizionali, è possibile unire una serie di diramazioni `if-else` per ottenere l'effetto desiderato, come indicato di seguito:

```
if (x == 0)
  y = 5;
else if (x == 2)
  y = 25;
else if (x >= 3)
  y = 125;
```

In questo esempio, vengono effettuati tre diversi confronti, ognuno dei quali provoca l'esecuzione della relativa istruzione, se il risultato della condizione è `true`. Si noti tuttavia che le diverse diramazioni `if-else` in realtà sono annidate nella precedente. In questo modo si garantisce che venga eseguita almeno un'istruzione.

L'ultimo argomento importante da discutere in merito alle diramazioni if-else è quello delle istruzioni composte. Come indicato nella nota precedente, un'istruzione composta è un blocco di codice incluso tra parentesi graffe che si presenta a un blocco esterno come un'unica istruzione. Di seguito è riportato un esempio di istruzione composta utilizzato con una diramazione if:

```
if (eseguiCalcolo) {
  x += y * 5;
  y -= 10;
  z = (x - 3) / y;
}
```

A volte, quando si annidano diramazioni if-else, è necessario utilizzare le parentesi graffe per distinguere le istruzioni collegate a una diramazione. Il seguente esempio evidenzia il problema:

```
if (x != 0)
  if (y < 10)
    z = 5;
else
  z = 7;
```

In questo esempio, lo stile dei rientri indica che la diramazione else appartiene alla prima if (quella esterna). Tuttavia, poiché non è stato specificato un raggruppamento, il compilatore Java assume che else sia collegata alla diramazione if interna. Per ottenere il risultato desiderato, è necessario modificare il codice come segue:

```
if (x != 0) {
  if (y < 10)
    z = 5;
}
else
  z = 7;
```

L'aggiunta delle parentesi graffe indica al compilatore che l'istruzione if interna fa parte di un'istruzione composta e, cosa ancora più importante, permette di nascondere completamente la diramazione else dall'istruzione if interna. Sulla base di ciò che si è appreso dalla discussione sui blocchi nel Capitolo 2, è possibile capire che il codice all'interno della diramazione if interna non è in grado di accedere al codice all'esterno del suo ambito, inclusa la diramazione else.

Il Listato 3.11 contiene il codice sorgente della classe NomeIfElse, che utilizza molti degli elementi discussi finora.

Listato 3.11 *La classe NomeIfElse.*

```
class NomeIfElse {
  public static void main (String args[]) {
    char inizialeNome = (char)-1;
    System.out.println("Immettere l'iniziale del nome:");
    try {
      inizialeNome = (char)System.in.read();
    }
```

```
      catch (Exception e) {
        System.out.println("Errore: " + e.toString());
      }
      if (inizialeNome == -1)
        System.out.println("Che razza di nome è questo?");
      else if (inizialeNome == 'g')
        System.out.println("Il nome deve essere Giulio!");
      else if (inizialeNome == 'v')
        System.out.println("Il nome deve essere Vincenzo!");
      else if (inizialeNome == 'p')
        System.out.println("Il nome deve essere Paolo!");
      else
        System.out.println("Non riesco a indovinare il nome!");
    }
  }
```

Quando si digita v in risposta al messaggio di input, `NomeIfElse` produce il seguente risultato:

```
Il nome deve essere Vincenzo!
```

La cosa che probabilmente colpisce per prima in `NomeIfElse` è il metodo `read()`. Questo legge semplicemente un carattere dal dispositivo di input standard (`System.in`), che di norma è la tastiera. Si noti che viene utilizzato un casting, in quanto `read()` restituisce un tipo `int`. Dopo aver richiamato il carattere immesso, viene utilizzata una serie di diramazioni `if-else` per determinare l'output appropriato. Se non vi sono corrispondenze, viene eseguita la diramazione `else` finale, che informa gli utenti che non è stato possibile determinare il loro nome. Si noti che viene controllato il valore di `read()` per verificare che sia uguale a –1; il metodo `read()` restituisce `-1` se ha raggiunto la fine del flusso di input.

La chiamata al metodo `read()` in `NomeIfElse` è inclusa in una clausola `try-catch`, che fa parte del supporto di Java per la gestione degli errori e che in questo caso viene utilizzata per identificare gli errori durante la lettura dell'input dell'utente. Le eccezioni e le clausole `try-catch` sono discusse nel Capitolo 6.

Diramazioni switch

Le diramazioni `switch`, simili alle diramazioni `if-else`, sono state pensate specificatamente per passare in modo condizionale tra diversi risultati. La sintassi di `switch` è:

```
switch (Espressione) {
  case Costante1:
    ElencoIstruzioni1
  case Costante1:
    ElencoIstruzioni2
  default:
    ElencoIstruzioniDefault
}
```

Le diramazioni `switch` valutano e confrontano *`Espressione`* con tutte le costanti `case` e passano l'esecuzione del programma all'elenco di istruzioni corrispondente. Se nessuna costante `case` corrisponde a *`Espressione`*, il programma passa a *`ElencoIstruzioniDefault`*, se

presente (è opzionale). Un *elenco di istruzioni* è semplicemente una serie di istruzioni. A differenza della diramazione `if-else`, che indirizza il flusso di programma verso un'istruzione semplice o composta, la diramazione `switch` lo indirizza verso una serie di istruzioni.

Quando l'esecuzione del programma passa a una serie di istruzioni `case`, prosegue da quel punto in maniera sequenziale. Per capire meglio questo procedimento, si osservi il Listato 3.12, che contiene una versione con `switch` del programma dei nomi sviluppato precedentemente con le diramazioni `if-else`.

Listato 3.12 *La classe SwitchNome1.*

```
class SwitchNome1 {
  public static void main (String args[]) {
    char inizialeNome = (char)-1;
    System.out.println("Immettere l'iniziale del nome:");
    try {
      inizialeNome = (char)System.in.read();
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
    switch(inizialeNome) {
      case (char)-1:
        System.out.println("Che razza di nome è questo?");
      case 'g':
        System.out.println("Il nome deve essere Giulio!");
      case 'v':
        System.out.println("Il nome deve essere Vincenzo!");
      case 'p':
        System.out.println("Il nome deve essere Paolo!");
      default:
        System.out.println("Non riesco a indovinare il nome!");
    }
  }
}
```

Quando si digita v in risposta al messaggio di input, `SwitchNome1` produce i seguenti risultati:

```
Il nome deve essere Vincenzo!
Il nome deve essere Paolo!
Non riesco a indovinare il nome!
```

Il risultato errato è causato dal modo in cui la diramazione `switch` controlla il flusso del programma. `switch` ha combinato la v immessa con la corretta istruzione `case`, come indicato nella prima riga stampata, tuttavia il programma ha continuato l'esecuzione di tutte le istruzioni `case` da quel punto in poi, cosa che non doveva accadere. La soluzione al problema viene fornita dall'istruzione `break`, che obbliga un programma a uscire dal blocco di codice in esecuzione.La nuova versione del programma nel Listato 3.13 include le istruzioni `break` nei punti appropriati.

Listato 3.13 *La classe SwitchNome2.*

```
class SwitchNome2 {
  public static void main (String args[]) {
    char inizialeNome = (char)-1;
    System.out.println("Immettere l'iniziale del nome:");
    try {
      inizialeNome = (char)System.in.read();
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
    switch(inizialeNome) {
      case (char)-1:
        System.out.println("Che razza di nome è questo?");
        break;
      case 'g':
        System.out.println("Il nome deve essere Giulio!");
        break;
      case 'v':
        System.out.println("Il nome deve essere Vincenzo!");
        break;
      case 'p':
        System.out.println("Il nome deve essere Paolo!");
        break;
      default:
        System.out.println("Non riesco a indovinare il nome!");
    }
  }
}
```

Quando si esegue `SwitchNome2` e si immette v, si ottiene il seguente output:

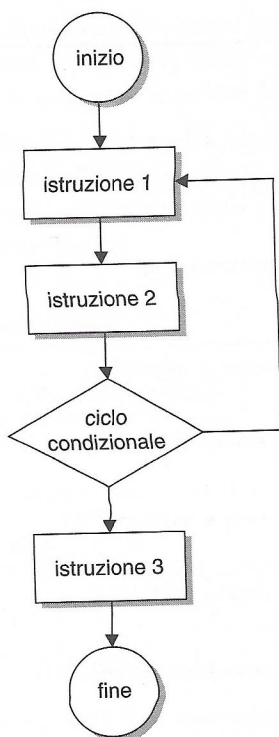
```
Il nome deve essere Vincenzo!
```

Come si può vedere, inserendo un'istruzione `break` dopo ogni istruzione `case`, si evita che il programma passi alle istruzioni `case` successive. Nonostante le istruzioni `break` vengano utilizzate in questo modo la maggior parte delle volte, vi possono essere situazioni in cui si desidera che l'esecuzione passi da un'istruzione `case` alla successiva.

Cicli

Quando si parla di flusso di programma, oltre che delle diramazioni è necessario parlare dei *cicli*, che, in parole semplici, permettono di eseguire ripetutamente il codice. In Java vi sono tre tipi di cicli: `for`, `while` e `do-while`.

Come le istruzioni, i cicli modificano il flusso sequenziale dei programmi. Nella Figura 3.3 è indicato il modo in cui un ciclo modifica il flusso sequenziale di un programma Java.

Figura 3.3
Un programma con un ciclo.

Cicli for

I cicli `for` costituiscono uno strumento per ripetere una sezione di codice per un determinato numero di volte. Sono strutturati in modo che la sezione di codice venga ripetuta fino a quando viene raggiunto un limite.

La sintassi dell'istruzione `for` è:

```
for (EspressioneInizializzazione; CondizioneCiclo; EspressioneIncremento)
  Istruzione
```

I cicli `for` ripetono le righe della *`Istruzione`* per il numero di volte determinato dalla *`EspressioneInizializzazione`*, dalla *`CondizioneCiclo`* e dalla *`EspressioneIncremento`*. La *`EspressioneInizializzazione`* viene utilizzata per inizializzare una variabile di controllo del ciclo, la *`CondizioneCiclo`* confronta la variabile di controllo del ciclo con un valore limite, mentre la *`EspressioneIncremento`* specifica il modo in cui la variabile di controllo deve essere modificata prima della successiva iterazione del ciclo. Il seguente esempio mostra come sia possibile utilizzare un ciclo `for` per stampare i numeri da 1 a 10:

```
for (int i = 1; i < 11; i++)
  System.out.println(i);
```

Per prima cosa, `i` viene dichiarato come intero. Il fatto che ciò avvenga all'interno del corpo del ciclo `for` potrebbe sembrare strano, ma è del tutto lecito. `i` viene inizializzata a 1 nella

parte `EspressioneInizializzazione` del ciclo `for`. Di seguito viene valutata l'espressione condizionale `i < 11` per vedere se il ciclo debba proseguire. A questo punto `i` è ancora uguale a 1, pertanto la `CondizioneCiclo` viene valutata come `true` e viene eseguita la `Istruzione` (il valore di `i` viene stampato sul dispositivo di output standard). Successivamente `i` viene incrementato nella parte `EspressioneIncremento` del ciclo `for` e il processo viene ripetuto valutando nuovamente la `CondizioneCiclo`. Si continua in questo modo finché la `CondizioneCiclo` viene valutata come `false`, vale a dire quando è uguale a 11 (dopo dieci iterazioni).

Nel Listato 3.14 è incluso il programma `ContaFor`, che indica come utilizzare un ciclo `for` per contare il numero di volte immesso dall'utente.

Listato 3.14 *La classe ContaFor.*

```
class ContaFor {
  public static void main (String args[]) {
    char input = (char)-1;
    int  numToCount;
    System.out.println("Immettere un numero da contare compreso tra 0 e 10:");
    try {
      input = (char)System.in.read();
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
    numToCount = Character.digit(input, 10);
    if ((numToCount > 0) && (numToCount < 10)) {
      for (int i = 1; i <= numToCount; i++)
        System.out.println(i);
    }
    else
      System.out.println("Il numero non è compreso tra 0 e 10!");
  }
}
```

Quando viene eseguito il programma `ContaFor` e viene immesso il numero `4`, si ottiene il seguente output:

```
1
2
3
4
```

`ContaFor` richiede prima all'utente di inserire un numero compreso tra `0` e `10`. Dalla tastiera viene letto un carattere utilizzando il metodo `read()` e il risultato viene memorizzato nella variabile carattere `input`. Viene quindi utilizzato il metodo statico `digit` della classe `Character` per convertire il carattere nella relativa rappresentazione in base 10. Questo valore viene memorizzato nella variabile intera `numToCount`, che viene controllata per verificare che sia compresa tra `0` e `10`. In caso positivo viene eseguito un ciclo `for` che conta da `1` a `numToCount`, stampando di volta in volta ogni numero. Se `numToCount` non rientra nell'intervallo valido, viene stampato un messaggio di errore.

Prima di proseguire, vi è un piccolo problema in ContaFor che potrebbe essere passato inosservato. Si provi a eseguire nuovamente il programma e a digitare un numero maggiore di 9. Che cosa succede al messaggio di errore? Il problema è che ForCount recepisce solo il primo carattere dell'input; così, se si digita **10**, il programma prende in considerazione solamente l'1, procedendo come se tutto fosse a posto. Non è necessario preoccuparsi di risolvere questo problema adesso, in quanto lo si analizzerà nuovamente quando si approfondirà l'argomento di input e output nel Capitolo 11.

Cicli while

Come i cicli for, i cicli while hanno una condizione che controlla l'esecuzione dell'istruzione del ciclo; a differenza dei primi, tuttavia, i cicli while non hanno espressioni di inizializzazione o di incremento. La sintassi è:

```
while (CondizioneCiclo)
  Istruzione
```

Se la *CondizioneCiclo* booleana viene valutata come true, viene eseguita la *Istruzione* e il processo inizia da capo. È importante capire che il ciclo while non ha espressioni di incremento come il ciclo for. Questo significa che il codice nella *Istruzione* deve in qualche modo influire sulla *CondizioneCiclo*, altrimenti il ciclo verrebbe ripetuto all'infinito, facendo in modo che un programma non venga mai terminato, bloccando il processore e talvolta anche il sistema.

Un altro aspetto importante da notare sui cicli while è che la *CondizioneCiclo* viene valutata prima del corpo della *Istruzione*. Ciò significa che se la *CondizioneCiclo* viene inizialmente valutata come false, la *Istruzione* non viene eseguita. Nonostante questo possa sembrare banale, in realtà è l'unica cosa che differenzia i cicli while dai cicli do-while, discussi nel prossimo paragrafo. Per capire meglio come funziona il ciclo while, si osservi il Listato 3.15, che mostra un programma di conteggio.

Listato 3.15 *La classe ContaWhile.*

```
class ContaWhile {
  public static void main (String args[]) {
    char input = (char)-1;
    int  numToCount;
    System.out.println("Immettere un numero compreso tra 0 e 10:");
    try {
      input = (char)System.in.read();
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
    numToCount = Character.digit(input, 10);
    if ((numToCount > 0) && (numToCount < 10)) {
      int i = 1;
      while (i <= numToCount) {
        System.out.println(i);
        i++;
```

```
            }
        }
        else
            System.out.println("Il numero non è compreso tra 0 e 10!");
    }
}
```

Il programma ContaWhile non mostra l'utilizzo migliore di un ciclo while. I cicli che prevedono un conteggio dovrebbero essere sempre implementati con for, ma vedendo il modo in cui è possibile creare un ciclo while che imita un ciclo for si può avere un'idea delle differenze strutturali tra i due tipi.

Poiché i cicli while non hanno alcun tipo di espressione di inizializzazione, prima di tutto è necessario dichiarare e inizializzare la variabile i a 1. Successivamente viene determinata la condizione per il ciclo while come i <= numToCount. All'interno dell'istruzione composta while vi è una chiamata al metodo println() , che stampa il valore di i. Infine, i viene incrementata e l'esecuzione del programma riprende dalla condizione del ciclo while.

Cicli do-while

I cicli do-while sono molto simili ai cicli while, come si può vedere nella seguente sintassi:

```
do
  Istruzione
while (CondizioneCiclo);
```

La differenza principale tra questi due tipi di cicli consiste nel fatto che nei cicli do-while la *CondizioneCiclo* viene valutata dopo che è stata eseguita la *Istruzione*. Questa differenza è importante, in quanto a volte si desidera che il codice della *Istruzione* venga eseguito almeno una volta, indipendentemente dalla *CondizioneCiclo*.

La *Istruzione* viene eseguita all'inizio e da questo punto in poi l'esecuzione continua finché la *CondizioneCiclo* viene valutata come true. Come con i cicli while, è necessario prestare attenzione a evitare di creare un ciclo infinito, che si ha quando la *CondizioneCiclo* rimane true all'infinito. Il seguente esempio mostra un ciclo do-while infinito:

```
do
  System.out.println("Sono bloccato!");
while (true);
```

Poiché la *CondizioneCiclo* è sempre true, viene sempre stampato il messaggio: "Sono bloccato!", per lo meno finché non si preme Ctrl+C e si esce dal programma.

Istruzioni break e continue

Si è già visto il funzionamento delle istruzioni break con le diramazioni switch. Le istruzioni break sono utili anche con i cicli: è possibile utilizzarle per uscire da un ciclo e saltare in modo efficace la condizione del ciclo stesso. Il Listato 3.16 mostra come un'istruzione break possa aiutare a uscire dal ciclo infinito visto precedentemente.

Listato 3.16 *La classe BreakCiclo.*

```
class BreakCiclo {
  public static void main (String args[]) {
    int i = 0;
    do {
      System.out.println("Sono bloccato!");
      i++;
      if (i > 100)
        break;
    }
    while (true);
  }
}
```

In BreakCiclo, impostando la condizione del ciclo su true, viene creato un ciclo do-while evidentemente infinito. L'istruzione break viene utilizzata per uscire dal ciclo quando i viene incrementato oltre 100.

Un'altra istruzione utile che funziona in modo simile a break è continue. A differenza della prima, questa risulta utile solo quando si lavora con i cicli e non può essere applicata alle diramazioni switch. L'istruzione continue funziona come break, nel senso che esce dall'iterazione corrente di un ciclo; la differenza è che con continue l'esecuzione del programma viene ripristinata dalla condizione della verifica del ciclo, mentre break fa uscire completamente dal ciclo. Si può utilizzare break quando si desidera uscire e terminare un ciclo e continue quando si desidera passare immediatamente all'iterazione successiva. Il seguente esempio mostra la differenza tra le istruzioni break e continue:

```
int i = 0;
while (i++ < 100) {
  System.out.println("Ciclo con i.");
  break;
  System.out.println("Per favore non stamparmi!");
}
int j = 0;
while (j++ < 100) {
  System.out.println("Ciclo con j!");
  continue;
  System.out.println("Per favore non stamparmi!");
}
```

In questo esempio, il primo ciclo viene interrotto a causa dell'istruzione break dopo aver stampato una volta il messaggio. Si noti che il secondo messaggio non viene stampato perché l'istruzione viene eseguita prima di raggiungerlo. Il secondo ciclo invece stampa il messaggio 100 volte. Il motivo è che l'istruzione continue permette al ciclo di proseguire le iterazioni. In questo caso, l'istruzione serve solo per passare al secondo messaggio, che non è stato ancora stampato.

Riepilogo

In questo capitolo sono stati discussi molti argomenti. Si è iniziato con le espressioni, per poi passare agli operatori, al loro funzionamento e al modo in cui influiscono su tutti i tipi di dati. Il tempo speso sugli operatori non verrà rimpianto: essi sono il fulcro di quasi tutte le espressioni matematiche o logiche di Java.

Si è quindi passati alle strutture di controllo per apprendere i diversi tipi di istruzioni e di cicli. Le istruzioni e i cicli costituiscono strumenti per modificare il flusso dei programmi e nella programmazione Java sono importanti quanto gli operatori.

Avendo chiari i concetti presentati in questo capitolo, si è pronti ad andare un po' più in profondità in Java, analizzando la programmazione orientata agli oggetti con le classi, i package e le interfacce.